Reinforcement Learning in SmashyRoad: A Comprehensive Technical Report

Author: Ahmed
Date: April 5, 2025
Project: Advanced Machine Learning — Reinforcement Learning
Environment: `SmashyRoadEnv` (Custom 2D Grid World)

Table of Contents

---

1. Project Overview

The `SmashyRoadEnv` is a 10x10 grid-world environment where: - The **agent** starts at `(2,2)` and must reach the goal at `(9,9)` - The **police** follows the agent greedily -

Obstacles block movement - **Reward**: `-1` per step, `-100` if caught, `+100` if goal reached

Environment Summary

Feature	Description
State Space	`(ax, ay, px, py)` — 10,000 states
Action Space	4 actions: Up, Down, Left, Right
Goal	Reach `(9,9)` without being caught
Max Steps	100

This report documents the **6-phase evolution** of RL agents trained on this environment.

---

2. Phase 1: Q-Learning — Model-Free Foundation

Objective

Train a **tabular Q-Learning agent** to learn the optimal policy through trial and error.

Implementation

```
 def q_learning(env, episodes=30000, alpha=0.1, gamma=0.99, epsilon_start=1.0,
epsilon_decay=2000):
    Q = np.zeros((10, 10, 10, 10, 4))  # State-action table
    for ep in range(episodes):
        state = env.reset()
        done = False
        epsilon = epsilon_start * np.exp(-ep / epsilon_decay)

        while not done:
            if np.random.rand() < epsilon:
                action = np.random.randint(0, 4)
            else:
                ax, ay, px, py = [int(x) for x in state]
                action = np.argmax(Q[ax, ay, px, py])

            next_state, reward, done = env.step(action)
            ax, ay, px, py = [int(x) for x in state]
            nax, nay, npx, npy = [int(x) for x in next_state]

            # Q-Learning update
            td_target = reward + gamma * np.max(Q[nax, nay, npx, npy]) if not
done else reward
            td_error = td_target - Q[ax, ay, px, py, action]
            Q[ax, ay, px, py, action] += alpha * td_error

            state = next_state
    return Q
```

Explanation

- **Q-Table**: Stores expected future reward for each (state, action)
- **Epsilon Decay**: Starts with exploration, shifts to exploitation
- **TD Update**: Adjusts Q-values based on observed reward and next-state max Q

Results

- **Final Win Rate**: 0.62
- **Learning Curve**: Slow start, steady improvement after 20,000 episodes

---

3. Phase 2: Value Iteration — Optimal Planning

Objective

Compute the optimal policy using dynamic programming.

Implementation

```
def value_iteration(env, gamma=0.99, theta=1e-6):
    V = np.zeros((10, 10, 10, 10))
    policy = {}

    while True:
        delta = 0
        for ax in range(10):
            for ay in range(10):
                for px in range(10):
                    for py in range(10):
                        s = (ax, ay, px, py)
                        v = V[ax, ay, px, py]
                        # Try all actions
                        action_values = []
                        for a in range(4):
                            env.agent_pos = [ax, ay]
                            env.police_pos = [px, py]
                            next_state, reward, done = env.step(a)
                            nax, nay, npx, npy = [int(x) for x in next_state]
                            action_values.append(reward + gamma * V[nax, nay,
npx, npy])
                        best_action = np.argmax(action_values)
                        V[ax, ay, px, py] = max(action_values)
                        policy[s] = best_action
                        delta = max(delta, abs(v - V[ax, ay, px, py]))
        if delta < theta:
            break
    return policy
```

Explanation

- **Value Iteration**: Solves Bellman optimality equation
- **Convergence**: Stops when value change < theta
- **Policy Extraction**: Greedy action at each state

Results

- **Win Rate**: 1.00

- **Iterations**: 176
- **Significance**: Proves environment is solvable and bug-free

---

4. Phase 3: Policy Gradient — The Variance Challenge

Objective

Learn a policy directly using gradient ascent on expected reward.

Attempt 1: Linear Policy

```
def policy_forward(state, W):
    logits = np.dot(state, W)
    logits -= np.max(logits)
    probs = np.exp(logits) / np.sum(np.exp(logits))
    return probs

# Gradient update
dW = np.outer(state, (action_onehot - probs) * advantage)
W += alpha * dW
```

Why It Failed

- **High Variance**: Monte Carlo returns are noisy
- **No Baseline**: No critic to reduce variance
- **Sparse Rewards**: No signal until goal is reached

Attempt 2: Actor-Critic

Added a value network to estimate V(s) and used advantage A(s,a) = Q(s,a) - V(s) — but still failed due to linear function approximation limitations.

---

5. Phase 4: Deep Q-Network (DQN)

Objective

Use deep learning to generalize across states.

Implementation (PyTorch)

```
class DQN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(4, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 4)
        )

    def forward(self, x):
        return self.net(x)
```

Training Loop

```
 loss = F.smooth_l1_loss(current_q, expected_q)
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()
```

Challenges

- `deepcopy(env)` was slow due to PyGame objects
- Gradient instability without clipping
- Simulation mismatch

Fix: Lightweight Environment

```
class LightweightSmashyRoadEnv:
    def __init__(self):
        self.agent_pos = [2, 2]
        self.police_pos = [7, 7]
        # No PyGame objects
```

Result: Win Rate 0.70

---

6. Phase 5: Rule-Based Agent

Objective

Use heuristic rules to win.

Implementation

```
def get_action(env):
    ax, ay = env.agent_pos
    px, py = env.police_pos

    if abs(ax - px) <= 1 and abs(ay - py) <= 1:
        # Escape logic
        ...
    else:
        # Move toward goal
        ...
```

Why It Failed

- Relied on `env.agent_pos`, which wasn't updated in simulation
- No fallback learning
- Brittle to edge cases

---

7. Phase 6: Monte Carlo Tree Search (MCTS)

Objective

Plan ahead by simulating possible futures.

Node Structure

```
class MCTSNode:
    def __init__(self, state):
        self.state = state
```

```
     self.children = []
    self.wins = 0
    self.visits = 0
    self.untried_actions = [0,1,2,3]

def ucb1(self):
    return self.wins / self.visits + 1.41 * np.sqrt(np.log(self.parent.visits)
/ self.visits)
```

```
### Selection & Expansion
```python
   while node.untried_actions == [] and node.children:
    node = max(node.children, key=lambda n: n.ucb1())
    temp_env.step(node.action)

if node.untried_actions:
    action = random.choice(node.untried_actions)
    node.untried_actions.remove(action)
    ...
```

Result: 0.00 win rate — simulation mismatch

---

8. Hybrid MCTS — The Breakthrough

Concept

Use Value Iteration policy to guide MCTS expansion.

Guided Expansion

```
 s = temp_env._get_state()
ax, ay, px, py = [int(round(x)) for x in s]
s_key = (ax, ay, px, py)

# Use optimal policy to choose action
action = value_policy.get(s_key)
if action is None or action not in node.untried_actions:
    action = random.choice(node.untried_actions)
else:
    action = int(action)
```
```

Explanation

- Bias search toward winning paths
- Reduces randomness in playouts
- Leverages optimal knowledge

Result: Win Rate 0.65

---

9. Final Results & Play Demo

Training Summary

Agent	Win Rate
Q-Learning	0.62
Value Iteration	1.00
Hybrid MCTS	0.65

Play Demo Output

```
    --- Playing: Hybrid MCTS Agent ---
Episode 1: Won in 18 steps!
Episode 2: Lost at step 17
Episode 3: Lost at step 17
```

Analysis

- Hybrid MCTS wins when it finds the optimal path
- Fails under tight police pressure
- Performance limited by number of simulations

---

10. Technical Challenges & Solutions

Challenge	Solution
`deepcopy(env)` too slow	Created `LightweightSmashyRoadEnv`
MCTS never wins	Used VI policy to guide expansion
Q-Learning slow	Increased exploration decay
Rule-based agent stuck	Switched to planning-based approach

---

11. Conclusion & Future Work

Conclusion

- Value Iteration is optimal and proves solvability
- Q-Learning learns well but not perfectly
- Hybrid MCTS combines planning and learning effectively
- Policy Gradient fails due to variance and linear limitations

Future Work

- Increase MCTS simulations to 50 or 100
- Add entropy bonus to encourage exploration
- Use PPO for stable deep policy learning
- Record videos of agent behavior
- Visualize Q-values as heatmaps

---

12. Appendices

Appendix A: Full Code Structure

- `env.py` : Environment logic
- `q_learning.py` : Tabular Q-Learning
- `value_iteration.py` : Optimal policy
- `train.py` : Training loop
- `play_agents.py` : Demo script

Appendix B: Hyperparameters

Parameter	Value
Episodes	30,000
Alpha (Q)	0.1
Gamma	0.99
Epsilon Decay	2000
MCTS Simulations	20

Appendix C: References

- Sutton & Barto, Reinforcement Learning: An Introduction
- PyTorch Documentation
- pygame.org (for environment rendering)